SEPARATING SCRIPT AND VOCABULARY EFFECTS IN MULTILINGUAL TOKENIZATION COST

Anonymous authors
Paper under double-blind review

ABSTRACT

Cross-lingual token cost depends both on a language's writing system and on how its tokenizer encodes that writing system. We show that token-count ratios on parallel text factor exactly into these two components, and measure them across seven production tokenizers, fifteen languages, and five script families on FLORES. This decomposition distinguishes languages that are expensive because their writing system requires more characters from those with inefficient tokenization. Devanagari and Arabic-script languages require two to three times as many tokens as English despite similar writing-system cost, with most of the excess attributable to the tokenizer (convention-dependent for Devanagari: 99.9% under code points, 76.9% under grapheme clusters, though the cost ratio is not). We then measure, in simulation, how much of this tokenizer penalty can be recovered by replacing unused entries with high-impact fragments at fixed vocabulary size. At 5,000 slots, mean characters-per-token improvement reaches 58.8% for Devanagari and 51.1% for Arabic-script languages (per-language means over the seven tokenizers, evaluated on held-out devtest), with a maximum of 159.9%. CJK languages have comparable tokenizer penalties, but a segmentation-free generator with supply well in excess of the budget recovers almost nothing more. These results show that multilingual tokenization disparities should first be diagnosed by cause: substantial inefficiency can often be recovered by reallocating existing vocabulary capacity, but fixed-vocabulary interventions have clear limits.

1 INTRODUCTION

Tokens are the unit of account for commercial language models. API calls are priced per token and context windows are measured in tokens, so a user whose language requires more tokens for the same content pays more and fits less into a fixed window.

Cross-lingual disparities in token counts are well documented. Petrov et al. (2023) report up to 15× disparity across FLORES-200, and Ahia et al. (2023) find a 5× pricing gap that correlates negatively with the Human Development Index of the affected communities. What this literature does not supply, and what this paper provides, is per-tokenizer, per-language measurement of recoverable headroom on deployed production systems: for each production tokenizer, how much of a language's cost a fixed-vocabulary intervention can remove.

These figures do not separate two distinct mechanisms that produce the same symptom. A writing system may require many characters to express a given amount of content, or a tokenizer may pack few characters into each token. The first is a property of the language, and no tokenizer change addresses it; the second is a design choice. Aggregate token counts confound them, and within a single language the two can point in opposite directions.

For a language ℓ and a reference language, here English, the cost of an aligned parallel sentence factors as

$$\underbrace{\frac{T_\ell}{T_{\text{en}}}}_{\text{cost ratio}} = \underbrace{\frac{C_\ell}{C_{\text{en}}}}_{\text{writing system}} \times \underbrace{\frac{\text{CPT}_{\text{en}}}{\text{CPT}_\ell}}_{\text{tokenizer}} \tag{1}$$

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

where T is tokens, C is characters and CPT is characters per token. We treat it as a measurement device rather than a result. Its use is that the two factors can be estimated separately for each tokenizer–language pair, and that they turn out to diverge.

The two terms diverge most sharply for Chinese. Chinese uses roughly a third of the characters English uses for the same FLORES sentence, which should make it inexpensive. It requires $1.14\times$ as many tokens as English (arithmetic mean over the seven tokenizers; Figure 1 reports the geometric mean, $1.13\times$). Its tokenizer term is $+1.208$ in logs against Hindi's $+0.973$, a difference of 0.235 in logs (a ratio of $1.26\times$), while the two languages' token bills differ by a factor of 2.6. The tokenizers serve both languages poorly; what separates the two bills is orthographic compactness. Ranking our fifteen languages by characters per token places Chinese 14th and Japanese 15th; ranking them by tokens per aligned sentence places Chinese 2nd and Japanese 8th.

We simulate vocabulary reallocation, replacing tokens unused on the evaluation corpus with high-frequency fragments the tokenizer currently splits, at constant vocabulary size and across budgets spanning two orders of magnitude.

Our contributions are:

1. We measure the writing-system and tokenizer components of cross-lingual cost for 105 tokenizer–language pairs, using an exact factorisation of the cost ratio (Equation 1) as the measurement device rather than as a finding (Section 4.1).
2. We measure how much of the tokenizer term reallocation recovers at fixed vocabulary size, finding that every script family responds and that the CJK residual persists even under a segmentation-free generator with supply well in excess of the budget, at budgets differing across tokenizers by more than an order of magnitude (Section 4.2).

2 Related Work

Measured disparities, unattributed causes. The cross-lingual token gap is well established (Petrov et al., 2023; Ahia et al., 2023; Teklehaymanot & Nejdl, 2025), and its consequences are not confined to cost: tokenizer quality tracks downstream performance about as strongly as pretraining data size (Rust et al., 2021), and poorly supported vocabulary items remain under-trained even in frontier models (Yang et al., 2024). What these measurements report is a single aggregate per language. A language may be expensive because its script needs many characters, or because the tokenizer packs few characters into each token, and the reported ratios are consistent with either. The distinction matters because only the second is a design choice a vendor can act on. Schmidt et al. (2024) caution that compression alone does not establish tokenizer quality and use characters per token to attribute cost, not to rank tokenizers.

Vocabulary allocation across languages. Zheng et al. (2021) study how vocabulary capacity should be allocated across languages in cross-lingual pretraining, and Liang et al. (2023) show that enlarging and rebalancing a shared multilingual vocabulary improves masked language models; Limisiewicz et al. (2023) find that allocation and cross-language overlap, rather than script, govern how well a shared vocabulary serves each language. Together these establish vocabulary allocation as the operative constraint, which we take as background rather than as a finding of this work. All three concern tokenizers the authors train. What they do not supply, and what we provide, is the same measurement for tokenizers already in production: how much of the tokenizer term is recoverable at fixed vocabulary size.

Modifying a fixed vocabulary. Our intervention edits a vocabulary rather than trains one, which places it alongside a smaller line of work. Liu et al. (2023) build a task-specific vocabulary and merge it into an existing one; Liu et al. (2024) select expansion candidates adaptively for domain-specific models. Both confront the candidate-selection problem our impact-ranked heuristic faces, and we compare against simpler rules in Appendix C. Language-specific extensions show the gains are real — Cui et al. (2023) add 20K Chinese tokens to LLaMA and Hong et al. (2024) report a $1.7\times$ speedup — but both enlarge the vocabulary and so require embedding initialisation (Minixhofer et al., 2022), which we do not perform. We hold vocabulary size constant and report token counts, not task performance.

A different response is to drop the fixed vocabulary altogether, whether by operating on bytes or characters (Clark et al., 2022; Xue et al., 2022) or by learning patch boundaries dynamically (Pagnoni et al., 2025). Our results bear on when that cost is worth paying: reallocation recovers a substantial share of the tokenizer term for every script family we test, but leaves a CJK residual that no fixed vocabulary resolves (Section 4.2).

3 METHOD

3.1 DATA AND TOKENIZERS

We use FLORES (Goyal et al., 2022). All baseline quantities — character counts, token counts and the decomposition — are computed on the dev split ($n = 997$ sentences per language). The reallocation experiment selects replacement fragments on dev and evaluates the frozen maps on the disjoint devtest split ($n = 1{,}012$), so selection and evaluation never share sentences. FLORES sentences are translations of the same source, so differences in token count between languages reflect encoding rather than content.

Languages. We cover 15 languages across five script families: Latin (en, es, fr, de, it, pt), CJK (zh, ja, ko), Devanagari (hi, ne), Arabic (ar, fa) and Cyrillic (ru, uk). Production tokenizers already allocate substantial vocabulary to all of these. These languages already have vocabulary, so what varies is how well it's allocated, not whether it exists. Lower-resource languages, where outright absence dominates instead, are analyzed at scale by Arnett et al. (2025).

Tokenizers. We evaluate seven widely deployed production multilingual tokenizers spanning the major open- and closed-weight families: GPT-5 (`o200k_base`, shared with GPT-4o; 200K), Gemma-3 (262K), Llama-4 (200K), Qwen-3 (152K), DeepSeek-V3 (128K), Hunyuan (128K) and Mistral-Large (33K). All are openly released and were retrieved on 10 March 2026. Appendix A gives exact identifiers, loading parameters and frameworks, and notes two naming caveats: `o200k_base` is not a GPT-5-specific vocabulary, and the Mistral-Large-2512 entries in our data tables reuse the Mistral-Large-Instruct-2411 tokenizer. Character counts and token IDs use `add_special_tokens=False`.

This gives 105 tokenizer–language cells, each summarising 997 dev sentences.

3.2 THE COST IDENTITY

Our primary quantity is the cost ratio: tokens required for an aligned FLORES sentence in language ℓ divided by tokens required for the same sentence in English, aggregated over the corpus. Equation 1 factors it into a writing-system term, the characters needed relative to English, and a tokenizer term, English characters per token relative to ℓ's.

Characters per token is computed in aggregate form, $\mathrm{CPT}_{\mathrm{agg}} = \sum_i C_i / \sum_i T_i$, since the identity cancels exactly only under this definition. Regressions use $\log \mathrm{CPT}_{\mathrm{agg}}$, which makes the factorisation additive and the two terms comparable in magnitude. Reallocation gains in Section 4.2 are reported as the mean of per-sentence ratios, following the rest of the intervention pipeline.

Characters here are Unicode code points, and the pipeline applies no Unicode normalisation. Code points are not semantically equivalent across scripts: a Devanagari syllable may span several code points and a Chinese character occupies one. The cost ratio is invariant to the character definition, to machine precision (8.9×10^{-16} across the 105 cells, verified in the committed recount), because it reduces algebraically to a token-count ratio; its division into a writing-system term and a tokenizer term is not, for scripts with combining sequences. Where the two conventions differ we report both (Section 4.1, Table 5); where they do not, the tokenizer term is convention-independent.

The tokenizer term is not a pure design property. A Chinese character carries more content than a Latin character, so no tokenizer can bring Chinese to English's characters per token; part of the term reflects the script's information density rather than tokenizer error. The term is therefore an upper bound on what a tokenizer change could recover.

3.3 THE REALLOCATION EXPERIMENT

To measure how much of the tokenizer term is removable, we simulate a vocabulary change that adds no parameters. For each cell we identify words the tokenizer splits into at least two tokens, rank the resulting fragments by impact (frequency $\times$ tokens saved) and take the top k. Gains are reported at a budget of $k = 5{,}000$; the two-or-more rule is primary throughout, and the budget sweep of Figure 3 uses the two-or-more pool. Each selected fragment is added and one existing token removed, holding vocabulary size exactly constant. Removal candidates are restricted to tokens with zero observed frequency on the development corpus. The wrapper matches added fragments ahead of an unmodified base tokenizer, so a displaced entry is still emitted for its own string and its removal costs nothing in token count; vocabulary-size invariance is an accounting property of the simulation, not a demonstrated property of a retrained tokenizer. The claim this supports is narrow: coverage can be added without new embedding parameters. Measurement per cell is deterministic, so the ranges and IQRs reported in Section 4.2 describe spread across tokenizers or languages, not sampling uncertainty.

We compare this against augmentation, which adds the same fragments without removing anything and therefore serves as an upper bound. If reallocation matches augmentation, the displaced tokens were contributing nothing measurable to tokenization efficiency.

Two implementation details matter for reproduction. First, most of these tokenizers merge a leading-space marker onto word-initial tokens, so candidate fragments must be stored with the marker convention appropriate to each framework; storing bare words instead causes roughly 70% of insertion events to be silently rejected at match time, and the rejection rate varies by script, which distorts cross-family comparison rather than simply attenuating it. Second, encoding under the modified vocabulary uses greedy longest-match on added fragments with base-tokenizer fallback. Across the 105 cells evaluated, no sentence's token count increased; greedy longest-match over a BPE base is not monotone in general, so this is an observation, not a guarantee. Appendix B documents per-framework handling of BPE merge ranks, SentencePiece unigram probabilities, and Tekken and tiktoken byte fallback.

Allocation is forced onto a single language per tokenizer, which measures per-language headroom rather than a deployable multi-language configuration. A deployable multi-language configuration would divide the same budget across languages, and the per-language figures we report are therefore an upper bound on what any single joint allocation delivers. Appendix E evaluates that joint allocation on a disjoint corpus.

4 RESULTS

4.1 THE COST OF THE TOKENIZER VS THE SCRIPT

Figure 1 decomposes each language's cost into the two terms of Equation 1. The tokenizer term dominates almost everywhere: cost cannot be attributed to the script alone. Devanagari and Arabic-script languages have script efficiency similar to English's and a comparatively high tokenizer cost. Two scripts break the pattern. Chinese and Japanese carry the longest blue segments in the figure: their scripts save roughly a third and under a half of English's characters respectively, and yet they are only 1.13$\times$ and 1.54$\times$ more expensive than English. Their tokenizers are among the worst in the set, yet Chinese's total bill stays below Spanish's, because its writing system saves more than its tokenizer loses: the two terms point in opposite directions and nearly cancel.

This is why compression alone misidentifies which languages are underserved. Characters per token predicts the cost ratio only moderately ($r = -0.608$, $\rho = -0.518$), and the disagreements are not marginal: Chinese ranks 14th of 15 on characters per token and 2nd on cost, Japanese 15th and 8th. A vendor ranking languages by compression and a user reading an invoice would name different languages as the problem.

The right panel gives the one caveat that bites. For scripts with combining sequences the tokenizer's share of the gap depends on whether characters are counted as code points or grapheme clusters, and for Devanagari the two conventions differ by about twenty points, enough to reorder languages: Hindi's tokenizer term overtakes Chinese's under grapheme clusters, because Devanagari has combining sequences and Chinese none — only Hindi's term moves. The cost ratio itself is invariant to

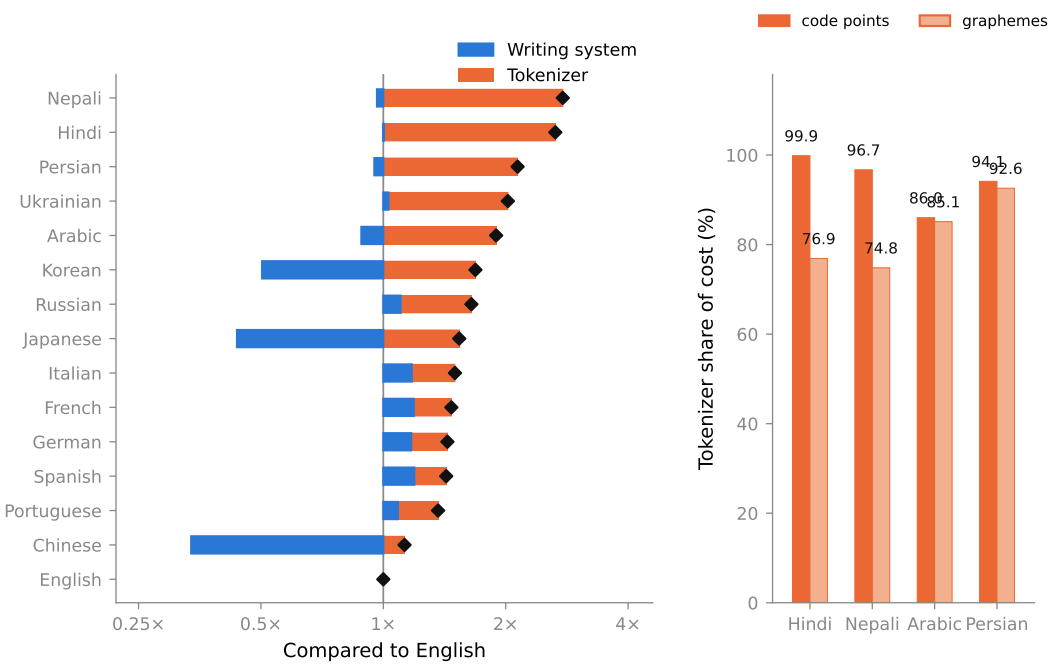


Figure 1: Cost of each language relative to English for the same FLORES sentence, decomposed by Equation 1 under the mean-of-logs convention: the writing-system term (blue) and tokenizer term (orange) are means of the per-tokenizer log terms, and diamonds mark their product, which is what the user pays. The figure's levels are therefore geometric means over the seven tokenizers, and are smaller than the arithmetic means of the per-tokenizer ratios reported in Table 5. For Devanagari and Arabic-script languages the cost is attributable almost entirely to the tokenizer, and for Cyrillic predominantly so. Chinese's writing system saves about 3× and its tokenizer returns nearly all of that saving. The right panel shows the tokenizer term's share of the total absolute log movement under code-point and grapheme-cluster counting for the four scripts with combining sequences: the split is convention-dependent (Hindi 99.9% to 76.9%, Nepali 96.7% to 74.8%), while the total cost ratio is unchanged. Table 5 reports arithmetic means of the per-tokenizer ratios.

this choice; only its division into terms is not. Arabic-script languages move little, and the remaining families not at all.

4.2 Marginal Impact, Not Supply, Limits CJK

Figure 2 plots every tokenizer–language pair: penalty on the x-axis, what reallocation recovers on the y. The dashed fit is the reference — if penalty were uniformly recoverable, the cells would track it.

Two families sit above the line. Devanagari and Arabic-script cells climb the upper right, several past 100%, which is to say reallocation more than doubles the characters those tokenizers fit into a token. Latin and Cyrillic cells sit in a tight band near the origin: small penalties, proportionate recovery, nothing surprising. English, which needs no reallocation, gains 4.0% — the floor that adding high-frequency fragments produces for any language, and the baseline against which the Devanagari and Arabic-script figures should be read.

The CJK cells are the story. They occupy the far right of the plot — the largest penalties in the set — and sit low, well beneath the fit and level with cells carrying a third of their penalty. Under the word-based generator, 20 of the 21 are supply-capped: the candidate pool runs out before the budget does (Table 2). Read alone, the ring suggests CJK stops improving because it runs out of fragments. The suggestion is wrong, and testing it is what pins down the mechanism.

Replacing the word generator with a segmentation-free one rules supply out: character n-grams of length 2–6, ranked by the same impact score under the same budget and dev-to-devtest protocol,

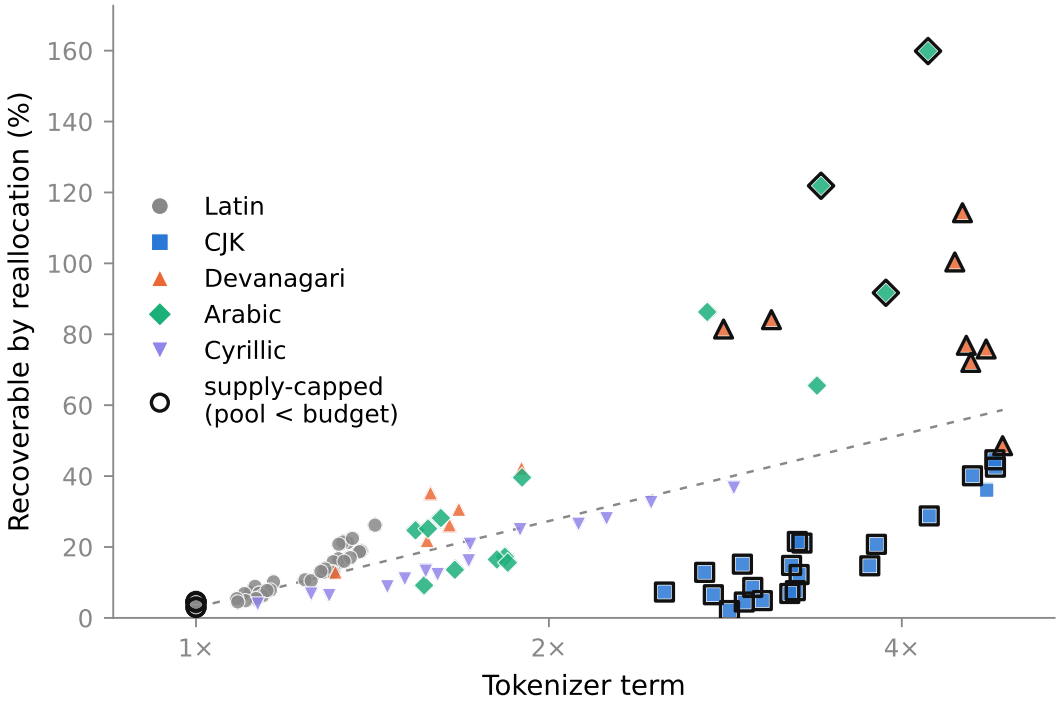


Figure 2: Which languages respond to vocabulary reallocation. Each point is one tokenizer–language pair under the two-or-more-token candidate rule at a 5,000-slot budget. The x-axis is the tokenizer term for that pair, the portion of the language's cost attributable to the tokenizer rather than to the writing system (Figure 1); the y-axis is the percentage improvement in characters per token. Cells whose candidate pool is smaller than the budget (supply-capped) are ringed. Devanagari and Arabic-script languages show large penalties that are largely recoverable. CJK penalties are comparable, and its family mean (17.8%) is comparable to Cyrillic's; 20 of its 21 cells exhaust the word-based pool at this rule (95.2%, Table 2), but the segmentation-free check (Section 4.2) shows supply is not the limit. The highest point (159.9%, Persian under Mistral-Large) is a tokenizer with almost no Persian coverage, where reallocation more than doubles characters per token. Dashed line: pooled fit.

leave every CJK cell with supply well in excess of the budget — none exhausts — and the gains barely move: Chinese falls 3.95 points, Japanese gains 4.55, Korean 6.00. The problem is not that fragments run out; it is that everything past the high-impact core is worthless. Reallocation at fixed vocabulary size is the wrong instrument for CJK, and the case for byte- and patch-level approaches rests on exactly this residual.

The single highest point, Persian under Mistral-Large at 159.9%, is a 33K vocabulary with almost no Persian coverage: its baseline of 1.03 characters per token is barely above character-level encoding. It is an extreme case of the same mechanism, not a separate one, and it is capped under the word-based rule, like the CJK cells beside it.

The headline gains are already out of sample: fragments are selected on FLORES dev and evaluated on the disjoint devtest split (Section 3.1). A separate 80/20 split within dev, which measures the in-sample advantage directly, puts the discount at eight points — mean gain over all 105 cells falls from 59.9% on selection sentences to 51.8% held out (Appendix D) — and frozen maps retain a median 96.9% of their FLORES gain on the disjoint NTREX-128 corpus (Appendix E).

4.3 Every Tokenizer Responds; They Differ in What It Costs

Figure 3 traces gain against budget for each of the seven. The curves separate into two groups by height, but the shape is the same throughout: every line rises, every line flattens, and every line crosses 5%. What differs is where. Mistral-Large and Hunyuan clear the threshold with 50 slots;

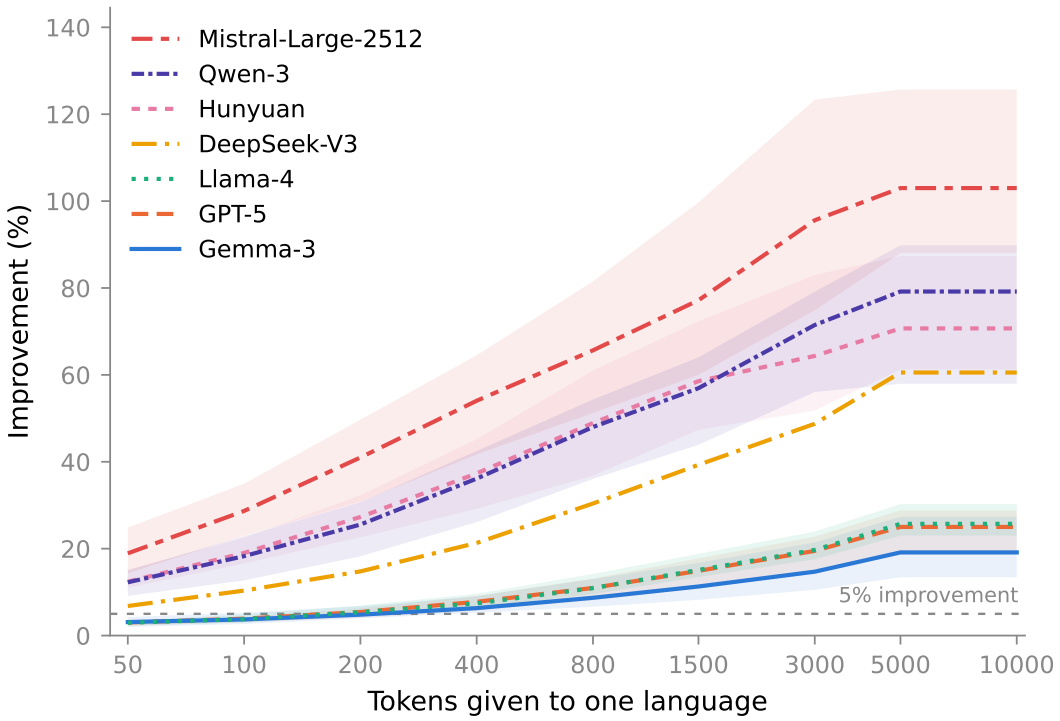


Figure 3: Improvement in characters per token as a function of reallocation budget, by tokenizer, under the two-or-more candidate rule, across budgets from 50 to 10,000 slots. Each line is the median over the four languages whose penalty is recoverable (Hindi, Nepali, Arabic, Persian); shaded bands are the interquartile range over those four languages, not confidence intervals. Budgets of 8,000 and 10,000 exhaust the word-based candidate pool and reproduce the 5,000-slot map exactly; the curves flatten there because the word-based pool is spent, and enlarging it recovers little (Section 4.2). All seven tokenizers cross the 5% line, four of them by a budget of 50 slots, on this median-over-four metric; the mean gain over all fifteen languages reaches 5% at larger budgets for five of the seven tokenizers (Table 4).

GPT-5, Gemma-3 and Llama-4 need 800. More than an order of magnitude separates the cheapest intervention from the most expensive, and none of the seven is a non-responder.

The flattening at the right of the plot is the same word-based pool exhaustion Figure 2 shows cell by cell: budgets of 8,000 and 10,000 reproduce the 5,000-slot map exactly because the word-based pool is exhausted, and enlarging it recovers little (Section 4.2). The curves stop at different heights, but they stop for one reason.

Within each script family the penalty predicts the gain closely ($r = 0.77$ to 0.97; partial correlation controlling for family, 0.817). Pooled across families it does not ($r = 0.543$), and CJK is what pulls it down — the cells that sit below the fit in Figure 2 are the same ones that break the pooled relationship.

One caution for anyone reproducing this. The two-group appearance of Figure 3 is a difference in budget, not in kind, and it is sensitive to how candidate fragments are stored. Omitting the leading-space marker of Section 3.3 suppresses recovery unevenly across tokenizers and manufactures a genuine-looking split into responders and non-responders. The effect is systematic, not noisy.

5 Discussion

Our results show that tokenization disparity and language cost disparity are related, but they are not the same problem. Cross-lingual differences in token counts are well established (Petrov et al., 2023). These differences also translate into real differences in API cost (Ahia et al., 2023). But

aggregate token counts cannot tell us whether a language is expensive because its writing system requires more characters or because its tokenizer encodes those characters inefficiently. Our decomposition separates these two effects and shows that they can point in opposite directions.

Chinese is the clearest example. It has one of the worst characters-per-token values in our study, but its compact writing system means that it still requires relatively few characters to express the same content. As a result, its total token cost is much lower than its compression score alone would suggest. This is important when interpreting claims that some languages are inherently more efficient for use with language models. Chinese does have a representation advantage, but poor tokenization prevents it from fully translating that advantage into lower token counts. More generally, characters per token is useful for describing tokenizer behaviour, but it should not be treated as a direct measure of language cost or tokenizer quality (Schmidt et al., 2024).

This distinction changes which languages appear to need intervention. Ranking our languages by compression identifies CJK as the main problem, whereas ranking them by token cost identifies Devanagari, Arabic-script, and Cyrillic languages. Neither ranking is wrong. They measure different sources of disparity. For tokenizer developers, the more useful first question is therefore not which languages have the worst token counts, but what is causing those counts.

Our reallocation results suggest that this diagnosis can lead to a simple fix. Prior work shows that how vocabulary capacity is distributed across languages affects the quality of multilingual tokenizers and models (Zheng et al., 2021). Liang et al. (2023) likewise show that rebalancing a multilingual vocabulary can improve performance without treating vocabulary size as the only constraint. Arnett et al. (2025) further show, across a large collection of monolingual tokenizers, that cross-lingual compression differences persist even when vocabulary size is held constant, and that uniformly increasing vocabulary size does not remove them. Our results extend this line of work to deployed multilingual tokenizers by asking a different question: how much of the existing tokenizer penalty can be recovered by reallocating the vocabulary without increasing its size? The answer is substantial for several language families.

The size of the required intervention is also notable. Several tokenizers show measurable gains with only 50 replacement slots. This suggests that much of the recoverable inefficiency is concentrated in a relatively small number of high-impact fragments. The practical implication is straightforward: before enlarging a vocabulary or rebuilding a tokenizer, developers can measure which languages have large tokenizer penalties, identify unused capacity, and test whether reallocating that capacity removes part of the gap. Existing methods for task-adaptive vocabulary construction similarly modify an existing vocabulary rather than treating the original allocation as fixed (Liu et al., 2023).

CJK shows where this strategy reaches its limit. These languages have large tokenizer penalties, and under the word-based generator most of their cells exhaust the candidate pool before the budget is reached. A segmentation-free generator with supply well in excess of the budget recovers almost nothing more: the fragments past the high-impact core carry little recoverable gain. The problem is therefore not simply that useful entries have been allocated to the wrong languages, nor that candidates run out. Under our fixed subword representation, the marginal fragments are worthless. This makes CJK particularly useful for interpreting the decomposition: it is the clearest case where poor tokenization and high user cost diverge, and where better vocabulary allocation is not enough.

This residual points toward approaches that do not rely on a fixed subword vocabulary. Byte- and character-level models avoid much of the dependence on learned vocabulary entries (Xue et al., 2022). Patch-based tokenization provides another way to reduce that dependence while retaining longer units (Pagnoni et al., 2025). Our experiments do not show that these approaches solve the CJK problem. They show why they become relevant once fixed-vocabulary reallocation has nothing high-impact left to add.

The main limitation is that our measure is deliberately narrower than downstream model quality. We estimate recoverable tokenization headroom under a fixed vocabulary and a particular candidate-generation procedure. Changing the vocabulary of a trained model may have effects beyond the token counts we measure, including effects that would require new embedding parameters (Minixhofer et al., 2022). The held-out and cross-corpus results indicate that the measured gains are not restricted to the selection corpus, but they do not establish a corresponding improvement in model performance. The appropriate interpretation is therefore not that a given percentage of token savings

will translate directly into better models, but that it identifies how much of an observed tokenizer penalty is recoverable by changing the vocabulary allocation.

Taken together, the results suggest a simple principle for multilingual tokenizer development: diagnose the source of the disparity before deciding how to fix it. Some languages have a large tokenizer problem that can be substantially reduced by reallocating existing vocabulary capacity. Others, particularly CJK in our experiments, reach the limits of what that intervention can provide. The value of the decomposition is precisely that it distinguishes these cases.

6 Conclusion

Cross-lingual token cost disparities have more than one cause, and separating those causes changes how they should be addressed. We show that the token cost of a language can be decomposed into a writing-system component and a tokenizer component, revealing cases where poor tokenization and high user cost point in different directions. This is especially clear for CJK languages: they are poorly compressed by production tokenizers, but their compact writing systems keep their overall token cost relatively low.

The same distinction helps identify where intervention is useful. Across seven production tokenizers, reallocation of existing vocabulary capacity substantially reduces tokenizer penalties for Devanagari, Arabic-script, and Cyrillic languages, in some cases with only 50 replacement slots. The gains show that a meaningful part of multilingual tokenization inefficiency can be addressed without increasing vocabulary size. CJK presents a different limit: a segmentation-free generator with supply well in excess of the budget recovers almost nothing beyond the word-based rule, leaving a residual that fixed-vocabulary reallocation cannot address.

Taken together, the results suggest a practical principle for multilingual tokenizer development: first diagnose the source of the disparity, then choose the intervention that matches it. Vocabulary reallocation is a simple and effective intervention when unused capacity is the constraint, but it is not a universal solution. Establishing how these vocabulary changes affect trained models is an important next step.

Reproducibility Statement

Appendix A lists the exact tokenizer identifiers, software versions, loading parameters, and retrieval dates. Appendix B specifies the reallocation procedure and candidate-generation details, and Appendices C–F report candidate-selection alternatives, the held-out split, cross-corpus transfer, and additional analyses. All code and data necessary to reproduce the experiments are provided as anonymized supplementary material.

AI Use Disclosure

In this work, we used generative AI tools to aid and polish the writing of the manuscript, for retrieval and discovery of related work, and to create and edit software code used in the analysis pipeline. We have not used generative AI tools to generate synthetic datasets, to develop theoretical models, to formulate or prove mathematical claims, to propose or refine hypotheses, to design the research methodology or experiments, to interpret results, or for translation or dataset construction. We have reviewed all AI-assisted work: AI-assisted code paths were executed to produce the reported results, AI-suggested related work was verified against the original sources, and all AI-edited text was checked by the authors. We take full responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

References

Orevaoghene Ahia, Sachin Kumar, Hila Gonen, Jungo Kasai, David R. Mortensen, Noah A. Smith, and Yulia Tsvetkov. Do all languages cost the same? Tokenization in the era of commercial language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Lan-*

guage Processing (EMNLP), pp. 9904–9923, Singapore, 2023. Association for Computational Linguistics. doi: 10.48550/arXiv.2305.13707. URL https://arxiv.org/abs/2305.13707.

Catherine Arnett, Tyler A. Chang, Stella Biderman, and Benjamin K. Bergen. Explaining and mitigating crosslingual tokenizer inequities. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025)*, 2025. doi: 10.48550/arXiv.2510.21909. URL https://arxiv.org/abs/2510.21909.

Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. CANINE: Pre-training an efficient tokenization-free encoder for language representation. *Transactions of the Association for Computational Linguistics*, 10:73–91, 2022. doi: 10.1162/tacl_a_00448.

Yiming Cui, Ziqing Yang, and Xin Yao. Efficient and effective text encoding for Chinese LLaMA and Alpaca. *arXiv preprint arXiv:2304.08177*, 2023. doi: 10.48550/arXiv.2304.08177. URL https://arxiv.org/abs/2304.08177.

Christian Federmann, Tom Kocmi, and Ying Xin. NTREX-128 – news test references for MT evaluation of 128 languages. In *Proceedings of the First Workshop on Scaling Up Multilingual Evaluation (SUMEval)*, pp. 21–24. Association for Computational Linguistics, 2022.

Naman Goyal, Cynthia Gao, Vishrav Chaudhary, Peng-Jen Chen, Guillaume Wenzek, Da Ju, Sanjana Krishnan, Marc'Aurelio Ranzato, Francisco Guzmán, and Angela Fan. The FLORES-101 evaluation benchmark for low-resource and multilingual machine translation. *Transactions of the Association for Computational Linguistics*, 10:522–538, 2022. doi: 10.1162/tacl_a_00474.

Jimin Hong, Gibbeum Lee, and Junkyu Cho. Accelerating multilingual language model for excessively tokenized languages. In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024. doi: 10.18653/v1/2024.findings-acl.660.

Davis Liang, Hila Gonen, Yuning Mao, Rui Hou, Naman Goyal, Marjan Ghazvininejad, Luke Zettlemoyer, and Madian Khabsa. XLM-V: Overcoming the vocabulary bottleneck in multilingual masked language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13142–13152. Association for Computational Linguistics, 2023.

Tomasz Limisiewicz, Jiří Balhar, and David Mareček. Tokenization impacts multilingual language modeling: Assessing vocabulary allocation and overlap across languages. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 5661–5681, Toronto, Canada, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.350.

Chengyuan Liu, Shihang Wang, Lizhi Qing, Kun Kuang, Yangyang Kang, Changlong Sun, and Fei Wu. Gold panning in vocabulary: An adaptive method for vocabulary expansion of domain-specific LLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 7442–7459. Association for Computational Linguistics, 2024.

Siyang Liu, Naihao Deng, Sahand Sabour, Yilin Jia, Minlie Huang, and Rada Mihalcea. Task-adaptive tokenization: Enhancing long-form text generation efficacy in mental health and beyond. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 15264–15281. Association for Computational Linguistics, 2023.

Benjamin Minixhofer, Fabian Paischer, and Navid Rekabsai. WECHSEL: Effective initialization of subword embeddings for cross-lingual transfer of monolingual language models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 3992–4006, Seattle, United States, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.293.

Artidoro Pagnoni, Ram Pasunuru, Pedro Rodriguez, John Nguyen, Benjamin Muller, Margaret Li, Chunting Zhou, Lili Yu, Jason Weston, Luke Zettlemoyer, Gargi Ghosh, Mike Lewis, Ari Holtzman, and Srinivasan Iyer. Byte latent transformer: Patches scale better than tokens. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pp. 9238–9258, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.453. URL https://arxiv.org/abs/2412.09871.

Aleksandar Petrov, Emanuele La Malfa, Philip H. S. Torr, and Adel Bibi. Language model tokenizers introduce unfairness between languages. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, pp. 36963–36990, 2023. doi: 10.48550/arXiv.2305.15425. URL https://arxiv.org/abs/2305.15425.

Phillip Rust, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. How good is your tokenizer? On the monolingual performance of multilingual language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pp. 3118–3135. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.acl-long.243.

Craig W. Schmidt, Varshini Reddy, Haoran Zhang, Alec Alameddine, Omri Uzan, Yuval Pinter, and Chris Tanner. Tokenization is more than compression. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 678–702. Association for Computational Linguistics, 2024.

Hailay Kidu Teklehaymanot and Wolfgang Nejdl. Tokenization disparities as infrastructure bias: How subword systems create inequities in LLM access and efficiency. *arXiv preprint arXiv:2510.12389*, 2025. doi: 10.1109/FLLM67465.2025.11390956. URL https://arxiv.org/abs/2510.12389.

Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. ByT5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 2022. doi: 10.1162/tacl_a_00461.

Jinhao Yang, Zhiying Wang, Yifan Lin, and Zhengyu Zhao. Problematic tokens: Tokenizer bias in large language models. In *Preprint*, 2024. doi: 10.1109/BigData62323.2024.10825615. URL https://arxiv.org/abs/2407.04572.

Bo Zheng, Li Dong, Shaohan Huang, Saksham Singhal, Wanxiang Che, Ting Liu, Xia Song, and Furu Wei. Allocating large vocabulary capacity for cross-lingual language model pre-training. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3203–3215. Association for Computational Linguistics, 2021.

A Reproducibility

Table 1 lists the exact HuggingFace identifier or tiktoken encoding name for each tokenizer, with its underlying framework. Open-weight tokenizers are downloaded at runtime via `AutoTokenizer.from_pretrained` with `use_fast=True` and `trust_remote_code=True`; the GPT-5 encoding is loaded via `tiktoken.get_encoding("o200k_base")` (tiktoken 0.12.0). All artefacts were retrieved on 10 March 2026. Character counts and token IDs use `add_special_tokens=False`.

Table 1: Tokenizer identifiers and frameworks.

Short name	Identifier	Framework
Gemma-3	google/gemma-3-270m-it	SentencePiece
Llama-4	meta-llama/ Llama-4-Maverick-17B-128E-Instruct	Tiktoken
Qwen-3	Qwen/Qwen3-235B-A22B	Tiktoken
DeepSeek-V3	deepseek-ai/DeepSeek-V3.2	HF Fast
Hunyuan	tencent/Hunyuan-7B-Instruct	Tiktoken
Mistral-Large	mistralai/ Mistral-Large-Instruct-2411	Tekken
GPT-5	o200k_base (tiktoken 0.12.0)	Tiktoken

OpenAI has not released a GPT-5-specific encoding; the public tiktoken library ships `o200k_base` for both GPT-4o and GPT-5, and we treat it as the tokenizer of record for that family. The label

should not be read as implying a separately trained vocabulary. Where data tables list Mistral-Large-2512, this is the December 2025 release, which reuses the Mistral-Large-Instruct-2411 tokenizer; we load the 2411 repository as canonical.

Vocabulary sizes are the raw values returned by `vocab_size`, or `n_vocab` for tiktoken, at the retrieval date, rounded to the nearest thousand in prose.

B Reallocation Pipeline

For each cell: tokenize the development corpus; identify words split into at least two tokens; score the resulting fragments by frequency $\times$ tokens saved; take the top k; identify the k vocabulary entries with zero observed frequency on the development corpus; pair each fragment with one such entry; encode the evaluation corpus under the modified vocabulary using greedy longest-match on added fragments with base-tokenizer fallback; and compare characters per token before and after.

Candidate identification and ranking. We tokenize the corpus, identify words split into at least two tokens, and rank candidates by impact $=$ frequency $\times(n_{\text{tokens}} - 1)$. Words are whitespace-delimited for space-delimited languages; Chinese, Japanese and Korean are segmented morphologically (jieba, fugashi or janome, and kiwipiepy, respectively). The robustness check in Section 4.2 replaces this generator with character n-grams of length 2–6 over word-like character runs (no morphological segmentation), filtered and ranked identically. The top k are selected, where k is the budget.

Leading-space markers. Candidate fragments are stored with the marker convention appropriate to each framework. This is not optional: most of these tokenizers merge a leading-space marker onto word-initial tokens, so a matcher that stores bare words silently rejects roughly 70% of insertion events at match time. The rejection rate varies by script — it is highest for Latin and Cyrillic and lowest for CJK — so the effect is not a uniform attenuation but a distortion of cross-family comparison, and it is sufficient to produce an apparent partition of tokenizers into responders and non-responders.

Per-framework handling. Added fragments are matched ahead of the base tokenizer rather than inserted into its merge table, so BPE merge ranks and SentencePiece unigram probabilities are left untouched and no rescoring is required. Fragments must carry the leading-space marker convention of their framework, since most of these tokenizers merge a space onto word-initial tokens. Tekken and tiktoken byte fallback is unaffected because unmatched spans fall through to the base tokenizer unchanged. A fragment identical to an existing vocabulary entry is skipped at selection time, so no entry is duplicated.

Encoding under the modified vocabulary. Fragments are applied by greedy longest-match with base-tokenizer fallback. Reported gains are computed from direct sentence-level encode lengths under the modified vocabulary, making them constructive lower bounds on achievable characters per token.

Vocabulary size invariance. At every budget level the number of tokens removed equals the number added.

C Candidate-Selection Alternatives

We compared five selection rules at a common budget of 5,000 across all 105 cells, fragments selected on dev and gains evaluated on devtest. Mean gain over the 28 Devanagari and Arabic-script cells: impact-ranked 54.9%, random displacement 54.9%, frequency-only fragment ranking 55.2%, tokens-saved-only 53.4%, script-balanced allocation 25.4%.

Two results follow. First, *which* entries are displaced makes no difference at all: impact-ranked and a seeded random displacement agree to machine precision on every cell, despite displacing different entries. Under this wrapper architecture the base tokenizer is unmodified, so a displaced entry is

Table 2: Share of cells that exhaust the candidate supply before the 5,000-slot budget is spent, by script family and candidate rule (fragments from words split into at least three, two or one tokens). A cell exhausts when the impact-ranked fragment pool is smaller than the budget.

Family	≥ 3	≥ 2	≥ 1
Latin	78.6%	11.9%	0%
Cyrillic	7.1%	0%	0%
Devanagari	78.6%	57.1%	57.1%
Arabic	57.1%	21.4%	21.4%
CJK	100%	95.2%	66.7%

still emitted for its own string and its removal costs nothing in token count. This is a property of the simulation, not evidence that removal is free in a retrained model.

Second, concentration beats balance. Script-balanced allocation divides the budget across families in proportion to language counts and fills all 5,000 slots, yet recovers less than half of what concentrating the same budget on one language's own fragments recovers.

D Held-Out Split

We split FLORES dev 80/20, select replacement maps on the larger part, freeze them, and evaluate on both parts. With maps selected on the full corpus, mean gain over all 105 cells is 59.9% on the selection sentences and 51.8% on the held-out sentences, and per-cell gains correlate at $r = 0.97$. Selecting on the 797-sentence subset instead drops held-out gain to 24.5% and the correlation to $r = 0.86$: fragments chosen from a small sample often do not recur in a 200-sentence evaluation. Since the reported results select over the full corpus, the first regime is the relevant one, and the gap between 59.9% and 51.8% is the in-sample advantage to discount.

This tests stability within the FLORES distribution; Appendix E tests transfer to a disjoint corpus.

E Cross-Corpus Transfer

The reallocation gains of Section 4.2 select fragments on FLORES dev and evaluate them on devtest — sentences unseen during selection, but from the same corpus and domain. Whether the gains transfer to a different corpus is therefore still open. To test this, we froze the budget-5,000 replacement maps — selected on FLORES dev, with no re-selection — and evaluated them unchanged on NTREX-128 (Federmann et al., 2022), a news-translation corpus with no sentence overlap with FLORES.

For the per-language maps that produce the headline figures, per-cell gains on the two corpora correlate at $r = 0.998$. Restricted to the 89 cells whose FLORES gain exceeds 1%, the NTREX gain retains a median 96.9% of the FLORES gain (quartiles 89.1% and 104.6%). The joint, vocabulary-preserving allocation transfers at $r = 0.997$; on its 26 cells with a FLORES gain above 1% the median retention is 92.1% (quartiles 80.0% and 97.8%, maximum 114.2%, Ukrainian under Qwen-3). On no sentence in either corpus did the modified vocabulary produce more tokens than the base tokenizer, and no cell regresses — an empirical regularity across these cells, not a property of the method.

A map selected on FLORES therefore keeps most of its measured benefit on disjoint text, which is what the reallocation argument requires.

F Additional Results

F.1 What the Change Costs

Reallocation and augmentation differ by a mean of 0.000130 percentage points, median zero. Displaced entries were unused on the development corpus and the base tokenizer stays in place, so the

Table 3: Recoverable gain at budget 5,000 by language under the two-or-more candidate rule. Penalty is the tokenizer term from Table 5, computed on FLORES dev. Gain is the mean over the seven tokenizers of the two-or-more-rule gain, evaluated on the disjoint devtest split with replacement maps frozen at selection time; Range and IQR are across the seven tokenizers. The final column is the smallest budget at which the language's mean gain over the seven tokenizers exceeds 5%. Candidates are drawn from words split into at least two tokens.

Language	Family	Penalty	Gain (%)	Range	IQR	Budget to 5%
Persian	Arabic	2.43	69.4	[24.7–159.9]	77.4	50
Hindi	Devanagari	3.00	61.9	[12.9–114.3]	66.9	50
Nepali	Devanagari	3.18	55.7	[30.6–84.1]	35.7	50
Arabic	Arabic	2.28	32.8	[9.2–91.7]	26.8	100
Korean	CJK	3.44	23.2	[7.3–44.6]	16.8	400
Ukrainian	Cyrillic	2.02	23.0	[11.0–36.6]	14.8	200
Japanese	CJK	3.58	19.2	[6.5–42.5]	14.3	200
Italian	Latin	1.28	14.0	[6.3–22.4]	9.5	800
French	Latin	1.24	13.0	[6.9–21.4]	7.7	800
Portuguese	Latin	1.26	12.9	[5.2–26.2]	9.9	800
German	Latin	1.23	12.8	[6.9–21.3]	8.0	800
Russian	Cyrillic	1.53	12.4	[3.9–28.0]	10.0	800
Chinese	CJK	3.40	11.0	[2.1–36.0]	6.5	800
Spanish	Latin	1.20	10.4	[4.5–20.8]	8.3	1500
English	Latin	1.00	4.0	[3.0–5.7]	1.5	—

Table 4: Per-tokenizer response, ordered by recoverable gain on the Devanagari and Arabic-script languages, under the two-or-more candidate rule. Penalty is the mean tokenizer term over the fourteen non-English languages, computed on FLORES dev. Recoverable gain at budget 5,000 is evaluated on FLORES devtest, reported over all fifteen languages, and separately over those four languages (Hindi, Nepali, Arabic, Persian), which are the Devanagari and Arabic-script families; the latter column is not an average over all non-CJK languages. Range and IQR for the All-15 column are across the fifteen languages within each tokenizer. The final column is the smallest budget at which the tokenizer's mean gain over the fifteen languages exceeds 5%.

Tokenizer	Vocab.	Penalty	All 15	Range	IQR	Deva. + Arab.	Budget to 5%
Mistral-Large	33K	2.99	48.6	[5.7–159.9]	39.4	110.7	50
Hunyuan	128K	2.85	35.8	[4.5–121.9]	27.8	78.0	50
Qwen-3	152K	2.41	29.6	[4.5–100.4]	13.1	68.6	100
DeepSeek-V3	128K	2.08	23.9	[2.1–84.1]	12.4	55.6	200
Llama-4	200K	1.70	12.1	[3.0–42.3]	7.9	27.5	800
GPT-5	200K	1.83	13.4	[3.0–30.6]	11.8	22.5	800
Gemma-3	262K	1.67	11.9	[4.7–35.3]	5.8	21.6	800

result reflects those constraints rather than establishing anything independently; displacing entries at random gives identical gains (Appendix C). The narrow claim it supports is the one we need: coverage can be added without growing the vocabulary, and therefore without adding embedding parameters.

Table 5: Per-language decomposition. Columns 2 and 3 multiply to column 6 exactly, up to display rounding (entries are rounded from unrounded means, so multiplying the displayed values can differ in the last digit): the writing-system term is constant across tokenizers, so averaging over the seven preserves the identity. The tokenizer-term Range and IQR are across the seven tokenizers. Columns 7 and 8 are the tokenizer term's share of the total absolute log movement under code-point and grapheme-cluster counting (mean-of-logs convention); the share differs between conventions only for scripts with combining sequences, where the grapheme shares are Hindi 76.9% and Nepali 74.8% (Devanagari), Arabic 85.1% and Persian 92.6% (Arabic script). Dollar figures are input cost at $5 per million tokens.

Language	Char. ratio	Tok. ratio	Tok. range	Tok. IQR	Cost ratio	Tok. share	Tok. share (g)	$/1M chars
Nepali	0.96	3.18	[1.59–4.88]	2.78	3.07	96.7%	74.8%	2.67
Hindi	1.00	3.00	[1.32–4.72]	2.86	3.01	99.9%	76.9%	2.41
Persian	0.95	2.43	[1.54–4.21]	1.47	2.31	94.1%	92.6%	2.19
Ukrainian	1.03	2.02	[1.51–2.88]	0.64	2.08	95.9%	95.9%	2.01
Arabic	0.88	2.28	[1.57–3.88]	0.88	2.02	86.0%	85.1%	2.12
Korean	0.50	3.44	[2.51–4.80]	1.12	1.73	63.8%	63.8%	3.40
Russian	1.10	1.53	[1.13–2.24]	0.38	1.69	80.3%	80.3%	1.53
Japanese	0.44	3.58	[2.76–4.81]	0.91	1.56	60.3%	60.3%	3.62
Italian	1.17	1.28	[1.14–1.38]	0.17	1.50	60.3%	60.3%	1.33
French	1.19	1.24	[1.13–1.33]	0.17	1.47	55.2%	55.2%	1.29
German	1.17	1.23	[1.10–1.35]	0.21	1.44	56.6%	56.6%	1.28
Spanish	1.19	1.20	[1.08–1.32]	0.19	1.43	50.8%	50.8%	1.25
Portuguese	1.09	1.26	[1.13–1.42]	0.21	1.37	73.3%	73.3%	1.31
Chinese	0.34	3.40	[2.85–4.73]	0.51	1.14	52.6%	52.6%	3.45
English	1.00	1.00	[1.00–1.00]	0.00	1.00	—	—	1.05